```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import gzip

# Function to read and display first n lines of a gzipped file
def peek_file(file_path: str, n_lines: int = 10):
    print(f"Reading first {n_lines} lines from {file_path}:")
    print("-" * 80)

    try:
        with gzip.open(file_path, 'rt') as f:
            for i, line in enumerate(f):
                if i < n_lines:
                    print(f"Line {i+1}: {line.strip()}")
                else:
                    break
    except Exception as e:
        print(f"Error reading file: {e}")

    print("-" * 80)
    print()

# Read from both files
train_path = '/content/drive/My Drive/CS646/Project/train.gz'
test_path = '/content/drive/My Drive/CS646/Project/test.gz'

print("TRAINING DATA:")
peek_file(train_path)

print("\nTEST DATA:")
peek_file(test_path)
```

```
TRAINING DATA:
Reading first 10 lines from /content/drive/My Drive/CS646/Project/train.gz:
--------------------------------------------------------------------------------
Line 1: 0       M       4       0
Line 2: 0       0       Q       0       10047345        3080290,4098689 50504886,4217515        9848058,1084315 50534229
Line 3: 0       108     C       0       50628761
Line 4: 0       1080    C       0       50628761
Line 5: 1       M       4       0
Line 6: 1       0       Q       0       2057953 1093007 12695453,1284095        20124473,2056277        60660113,4693531
Line 7: 2       M       27      0
Line 8: 2       0       Q       0       2113437 1148783 33204613,3248226        2053036,303607  5878776,770558  34660823
Line 9: 3       M       13      1
Line 10: 3      0       Q       0       5239394 2365113,2856206,2491775 16457319,1712204        35513272,3344594
--------------------------------------------------------------------------------


TEST DATA:
Reading first 10 lines from /content/drive/My Drive/CS646/Project/test.gz:
--------------------------------------------------------------------------------
Line 1: 34573630        M       28      15
Line 2: 34573630        0       Q       0       10509813        3140263,2771769,3809197 34175267,3279130        34171511
Line 3: 34573630        6       C       0       34175267
Line 4: 34573630        250     T       1       2338823 1255934,3591935,1687744,3416736,4342741 56916272,4504808
Line 5: 34573633        M       29      22
Line 6: 34573633        0       T       0       21033027        4732554,3309263 36604172,3409323        37384416,3453277
Line 7: 34573634        M       28      27
Line 8: 34573634        0       T       0       353552  339852  36731685,3415022        13641671,1398997        45796560
Line 9: 34573635        M       28      32
Line 10: 34573635       0       Q       0       8447254 2947180,4807111 44298735,3856435        41815016,3712040
--------------------------------------------------------------------------------
```

```
import gzip
import pandas as pd
from typing import Dict, List, Tuple
from collections import defaultdict

class YandexDataParser:
    def __init__(self):
        self.sessions = defaultdict(dict)
        self.queries = []
        self.clicks = []

    def parse_record(self, line: str) -> Dict:
        """Parse a single line of the dataset"""
        parts = line.strip().split()

        # First field is always SessionID
```

```
            base_record = {
                'SessionID': int(parts[0])
            }

            # Handle different record types
            if 'M' in parts:
                # Meta record
                m_index = parts.index('M')
                base_record.update({
                    'Type': 'M',
                    'TimePassed': int(parts[1]) if m_index > 1 else 0,
                    'MetaInfo1': int(parts[m_index + 1]) if len(parts) > m_index + 1 else None,
                    'MetaInfo2': int(parts[m_index + 2]) if len(parts) > m_index + 2 else None
                })

            elif 'Q' in parts:
                # Query record
                q_index = parts.index('Q')
                base_record.update({
                    'Type': 'Q',
                    'TimePassed': int(parts[1]),
                    'SERPID': int(parts[q_index + 1]),
                    'QueryID': int(parts[q_index + 2]),
                    'URLs': parts[q_index + 3:] if len(parts) > q_index + 3 else []
                })
                self.queries.append(base_record)

            elif 'C' in parts:
                # Click record
                c_index = parts.index('C')
                base_record.update({
                    'Type': 'C',
                    'TimePassed': int(parts[1]),
                    'SERPID': int(parts[c_index + 1]),
                    'URLID': int(parts[c_index + 2]) if len(parts) > c_index + 2 else None
                })
                self.clicks.append(base_record)

            elif 'T' in parts:
                # Term record
                t_index = parts.index('T')
                base_record.update({
                    'Type': 'T',
                    'TimePassed': int(parts[1]),
                    'SERPID': int(parts[t_index + 1]),
                    'Terms': parts[t_index + 2:] if len(parts) > t_index + 2 else []
                })

            return base_record

        def process_file(self, file_path: str, max_lines: int = None) -> Tuple[pd.DataFrame, pd.DataFrame]:
            """Process the entire file and return query and click DataFrames"""
            line_count = 0

            try:
                with gzip.open(file_path, 'rt') as f:
                    for line in f:
                        try:
                            self.parse_record(line)
                            line_count += 1

                            if max_lines and line_count >= max_lines:
                                break

                            if line_count % 100000 == 0:
                                print(f"Processed {line_count} lines...")
                        except Exception as e:
                            print(f"Error processing line {line_count}: {e}")
                            print(f"Line content: {line.strip()}")
                            continue

            except Exception as e:
                print(f"Error reading file: {e}")

            # Convert to DataFrames
            queries_df = pd.DataFrame(self.queries)
            clicks_df = pd.DataFrame(self.clicks)

            # Calculate dwell times for clicks if we have clicks
            if not clicks_df.empty:
                clicks_df = self.calculate_dwell_times(clicks_df)

            return queries_df, clicks_df
            return queries_df, clicks_df

        def calculate_dwell_times(self, clicks_df: pd.DataFrame) -> pd.DataFrame:
            """Calculate dwell times and relevance grades for clicks"""
            # Sort by session and time
            clicks_df = clicks_df.sort_values(['SessionID', 'TimePassed'])

            # Calculate time difference to next event
            clicks_df['DwellTime'] = clicks_df.groupby('SessionID')['TimePassed'].diff().shift(-1)

            # Fill NaN values with median dwell time
            median_dwell = clicks_df['DwellTime'].median()
            clicks_df['DwellTime'] = clicks_df['DwellTime'].fillna(median_dwell)

            # Calculate relevance grades
            clicks_df['RelevanceGrade'] = pd.cut(
                clicks_df['DwellTime'],
                bins=[-float('inf'), 50, 400, float('inf')],
                labels=[0, 1, 2]
            )

            return clicks_df

    def main():
        # File paths
        train_path = '/content/drive/My Drive/CS646/Project/train.gz'
        test_path = '/content/drive/My Drive/CS646/Project/test.gz'

        # Set sample size for initial testing
        sample_size = 1000000  # Start with 1M lines

        # Process training data
        print("Processing training data (sample)...")
        parser = YandexDataParser()
        train_queries, train_clicks = parser.process_file(train_path, max_lines=sample_size)

        # Process test data
        print("\nProcessing test data (sample)...")
        parser = YandexDataParser()  # Reset parser
        test_queries, test_clicks = parser.process_file(test_path, max_lines=sample_size)

        # Print summary statistics
        print("\nTraining Data Summary:")
        print(f"Number of queries: {len(train_queries)}")
        print(f"Number of clicks: {len(train_clicks)}")
        if not train_clicks.empty:
            print("\nRelevance Grade Distribution (Training):")
            print(train_clicks['RelevanceGrade'].value_counts(normalize=True))

        print("\nTest Data Summary:")
        print(f"Number of queries: {len(test_queries)}")
        print(f"Number of clicks: {len(test_clicks)}")

        return (train_queries, train_clicks), (test_queries, test_clicks)

    if __name__ == "__main__":
        (train_queries, train_clicks), (test_queries, test_clicks) = main()
```

```
Processing training data (sample)...
Processed 100000 lines...
Processed 200000 lines...
Processed 300000 lines...
Processed 400000 lines...
Processed 500000 lines...
Processed 600000 lines...
Processed 700000 lines...
Processed 800000 lines...
Processed 900000 lines...

Processing test data (sample)...
Processed 100000 lines...
Processed 200000 lines...
Processed 300000 lines...
Processed 400000 lines...
Processed 500000 lines...
Processed 600000 lines...
Processed 700000 lines...
Processed 800000 lines...
Processed 900000 lines...

Training Data Summary:
Number of queries: 395514
Number of clicks: 391930

Relevance Grade Distribution (Training):
```

```
RelevanceGrade
1    0.678032
2    0.188100
0    0.133868
Name: proportion, dtype: float64

Test Data Summary:
Number of queries: 245526
Number of clicks: 210153
```

```
import numpy as np
import pandas as pd
from typing import List, Dict
import math
from collections import defaultdict

class SearchModelBase:
    def __init__(self, max_position: int = 10):
        self.max_position = max_position
        self.position_biases = np.zeros(max_position)
        self.relevance_scores = {}

    def _find_url_position(self, url_id: int, urls_list: List[str]) -> int:
        """Find position of URL in the results list"""
        url_str = str(url_id)
        for pos, url in enumerate(urls_list):
            if url.split(',')[0] == url_str:
                return pos
        return -1

class PositionBasedModel(SearchModelBase):
    def train(self, queries_df: pd.DataFrame, clicks_df: pd.DataFrame, sample_size: int = 5000):
        sampled_queries = queries_df.sample(n=min(sample_size, len(queries_df)), random_state=42)
        unique_serpids = set(sampled_queries['SERPID'].unique())
        sampled_clicks = clicks_df[clicks_df['SERPID'].isin(unique_serpids)]

        # Initialize counters
        position_clicks = np.zeros(self.max_position)
        position_views = np.zeros(self.max_position)
        url_clicks = defaultdict(int)
        url_views = defaultdict(int)

        # Process queries
        for _, query in sampled_queries.iterrows():
            urls = query['URLs']
            serpid = query['SERPID']

            if isinstance(urls, str):
                urls = urls.split(',')
            elif not isinstance(urls, list):
                continue

            # Count views for each position
            for pos, url in enumerate(urls[:self.max_position]):
                position_views[pos] += 1
                url_views[url] += 1

            # Process clicks for this query
            query_clicks = sampled_clicks[sampled_clicks['SERPID'] == serpid]
            for _, click in query_clicks.iterrows():
                position = self._find_url_position(click['URLID'], urls)
                if 0 <= position < self.max_position:
                    position_clicks[position] += 1
                    url_clicks[str(click['URLID'])] += 1

        # Calculate position biases
        for pos in range(self.max_position):
            if position_views[pos] > 0:
                self.position_biases[pos] = position_clicks[pos] / position_views[pos]

        # Calculate relevance scores
        for url_id, views in url_views.items():
            if views > 0:
                self.relevance_scores[url_id] = url_clicks[url_id] / views

        return self.position_biases, self.relevance_scores

    def predict(self, position: int, url_id: str, query_id: int = None) -> float:
        """Predict click probability"""
        if position >= self.max_position or url_id not in self.relevance_scores:
            return 0.0
        return self.position_biases[position] * self.relevance_scores[url_id]
```

```
class CascadeModel(SearchModelBase):
    def train(self, queries_df: pd.DataFrame, clicks_df: pd.DataFrame, sample_size: int = 5000):
        sampled_queries = queries_df.sample(n=min(sample_size, len(queries_df)), random_state=42)
        unique_serpids = set(sampled_queries['SERPID'].unique())
        sampled_clicks = clicks_df[clicks_df['SERPID'].isin(unique_serpids)]

        # Initialize counters
        examination_probs = np.zeros(self.max_position)
        url_relevance = defaultdict(float)
        url_views = defaultdict(int)
        url_clicks = defaultdict(int)

        # Process queries
        for _, query in sampled_queries.iterrows():
            urls = query['URLs']
            serpid = query['SERPID']

            if isinstance(urls, str):
                urls = urls.split(',')
            elif not isinstance(urls, list):
                continue

            # Count views and clicks
            for pos, url in enumerate(urls[:self.max_position]):
                url_views[url] += 1

            # Process clicks for this query
            query_clicks = sampled_clicks[sampled_clicks['SERPID'] == serpid]
            for _, click in query_clicks.iterrows():
                position = self._find_url_position(click['URLID'], urls)
                if 0 <= position < self.max_position:
                    url_clicks[str(click['URLID'])] += 1

                    # Update examination probabilities
                    examination_probs[position] += 1

        # Calculate examination probabilities
        for pos in range(self.max_position):
            if url_views:
                examination_probs[pos] /= len(url_views)

        # Calculate relevance scores
        for url_id, views in url_views.items():
            if views > 0:
                self.relevance_scores[url_id] = url_clicks[url_id] / views

        self.position_biases = examination_probs
        return self.position_biases, self.relevance_scores

    def predict(self, position: int, url_id: str, query_id: int = None) -> float:
        """Predict click probability in cascade model"""
        if position >= self.max_position or url_id not in self.relevance_scores:
            return 0.0

        # Cascade probability calculation
        click_prob = 1.0
        for p in range(position + 1):
            click_prob *= (1 - self.position_biases[p] * self.relevance_scores[url_id])

        return 1 - click_prob

class EnhancedPBM(SearchModelBase):
    def __init__(self, max_position: int = 10):
        super().__init__(max_position)
        self.query_weights = {}

    def _compute_query_weight(self, query):
        """Compute query-specific weight"""
        return len(query.split()) / 10.0 if query else 1.0

    def train(self, queries_df: pd.DataFrame, clicks_df: pd.DataFrame, sample_size: int = 5000):
        sampled_queries = queries_df.sample(n=min(sample_size, len(queries_df)), random_state=42)
        unique_serpids = set(sampled_queries['SERPID'].unique())
        sampled_clicks = clicks_df[clicks_df['SERPID'].isin(unique_serpids)]

        # Initialize counters
        position_clicks = np.zeros(self.max_position)
        position_views = np.zeros(self.max_position)
        url_clicks = defaultdict(int)
        url_views = defaultdict(int)

        # Compute query weights
        for _, query in sampled_queries.iterrows():
```

```
        for _, query in sampled_queries.iterrows():
            query_text = query.get('query', '') if 'query' in query.index else ''
            self.query_weights[query['SERPID']] = self._compute_query_weight(query_text)

        # Process queries
        for _, query in sampled_queries.iterrows():
            urls = query['URLs']
            serpid = query['SERPID']

            # Get query-specific weight
            query_weight = self.query_weights.get(serpid, 1.0)

            if isinstance(urls, str):
                urls = urls.split(',')
            elif not isinstance(urls, list):
                continue

            # Count views for each position
            for pos, url in enumerate(urls[:self.max_position]):
                position_views[pos] += 1
                url_views[url] += 1

            # Process clicks for this query
            query_clicks = sampled_clicks[sampled_clicks['SERPID'] == serpid]
            for _, click in query_clicks.iterrows():
                position = self._find_url_position(click['URLID'], urls)
                if 0 <= position < self.max_position:
                    position_clicks[position] += 1
                    url_clicks[str(click['URLID'])] += 1

        # Calculate position biases (η_i)
        for pos in range(self.max_position):
            if position_views[pos] > 0:
                self.position_biases[pos] = position_clicks[pos] / position_views[pos]

        # Calculate modified relevance scores (R'_i)
        for url_id, views in url_views.items():
            if views > 0:
                relevance_score = url_clicks[url_id] / views
                self.relevance_scores[url_id] = relevance_score

        return self.position_biases, self.relevance_scores

    def predict(self, position: int, url_id: str, query_id: int = None) -> float:
        """Predict click probability"""
        if position >= self.max_position or url_id not in self.relevance_scores:
            return 0.0

        # Retrieve query-specific weight
        query_weight = self.query_weights.get(query_id, 1.0) if query_id is not None else 1.0

        return (
            self.position_biases[position] *
            query_weight *
            self.relevance_scores[url_id]
        )

class ModelEvaluation:
    @staticmethod
    def calculate_ctr(predicted_clicks: np.ndarray, actual_clicks: np.ndarray) -> np.ndarray:
        """Calculate Click-Through Rate (CTR)"""
        predicted_clicks = np.array(predicted_clicks)
        actual_clicks = np.array(actual_clicks)

        ctr = np.zeros_like(predicted_clicks)
        for pos in range(len(predicted_clicks)):
            if predicted_clicks[pos] > 0:
                ctr[pos] = actual_clicks[pos] / predicted_clicks[pos]
            else:
                ctr[pos] = 0

        return ctr

    @staticmethod
    def calculate_mse(predicted_clicks: np.ndarray, actual_clicks: np.ndarray) -> float:
        """Calculate Mean Squared Error (MSE)"""
        predicted_clicks = np.array(predicted_clicks)
        actual_clicks = np.array(actual_clicks)

        mse = np.mean((predicted_clicks - actual_clicks) ** 2)

        return mse

    @staticmethod
    @staticmethod
    def calculate_ndcg(predicted_ranks: List[int], actual_relevance: List[int], k: int = 10) -> float:
        """Calculate Normalized Discounted Cumulative Gain (nDCG)"""
        def dcg_at_k(r, k):
            r = r[:k]
            return sum((2**r[i] - 1) / math.log2(i + 2) for i in range(len(r)))

        def idcg_at_k(r, k):
            r = sorted(r, reverse=True)[:k]
            return sum((2**r[i] - 1) / math.log2(i + 2) for i in range(len(r)))

        predicted_ranks = predicted_ranks[:k]
        actual_relevance = actual_relevance[:k]

        dcg = dcg_at_k(actual_relevance, k)
        idcg = idcg_at_k(actual_relevance, k)

        ndcg = dcg / idcg if idcg > 0 else 0

        return ndcg

def evaluate_models(queries_df: pd.DataFrame, clicks_df: pd.DataFrame):
    """Evaluate different click models"""
    # Initialize models
    pbm_model = PositionBasedModel()
    cascade_model = CascadeModel()
    enhanced_pbm_model = EnhancedPBM()

    # Train models
    pbm_model.train(queries_df, clicks_df)
    cascade_model.train(queries_df, clicks_df)
    enhanced_pbm_model.train(queries_df, clicks_df)

    # Metrics to store
    metrics = {
        'PBM': {'CTR': [], 'MSE': [], 'nDCG': []},
        'Cascade': {'CTR': [], 'MSE': [], 'nDCG': []},
        'Enhanced PBM': {'CTR': [], 'MSE': [], 'nDCG': []}
    }

    # Process each query
    for _, query in queries_df.iterrows():
        serpid = query['SERPID']
        urls = query['URLs']

        # Get clicks for this query
        query_clicks = clicks_df[clicks_df['SERPID'] == serpid]

        # Prepare actual click occurrences
        actual_clicks = np.zeros(len(urls))
        for _, click in query_clicks.iterrows():
            click_pos = pbm_model._find_url_position(click['URLID'], urls)
            if 0 <= click_pos < len(urls):
                actual_clicks[click_pos] = 1

        # Predict for each model
        models = {
            'PBM': pbm_model,
            'Cascade': cascade_model,
            'Enhanced PBM': enhanced_pbm_model
        }

        for model_name, model in models.items():
            # Predict clicks
            predicted_clicks = np.zeros(len(urls))
            for pos, url in enumerate(urls):
                predicted_clicks[pos] = model.predict(pos, url, serpid)

            # Calculate metrics
            ctr = ModelEvaluation.calculate_ctr(predicted_clicks, actual_clicks)
            mse = ModelEvaluation.calculate_mse(predicted_clicks, actual_clicks)

            # Calculate nDCG (use relevance grades from clicks)
            relevance_grades = [
                clicks_df[(clicks_df['SERPID'] == serpid) & (clicks_df['URLID'] == int(url.split(',')[0]))]['RelevanceGrade']
                if len(clicks_df[(clicks_df['SERPID'] == serpid) & (clicks_df['URLID'] == int(url.split(',')[0]))]) > 0
                else 0
                for url in urls
            ]

            ndcg = ModelEvaluation.calculate_ndcg(
                list(range(len(urls))),  # Predicted ranks
                relevance_grades
            )

            # Store metrics
            metrics[model_name]['CTR'].append(ctr)
            metrics[model_name]['MSE'].append(mse)
            metrics[model_name]['nDCG'].append(ndcg)

    # Compute average metrics
    for model_name in metrics:
        metrics[model_name]['CTR'] = np.mean(metrics[model_name]['CTR'], axis=0)
        metrics[model_name]['MSE'] = np.mean(metrics[model_name]['MSE'])
        metrics[model_name]['nDCG'] = np.mean(metrics[model_name]['nDCG'])

    return metrics

evaluation_results = evaluate_models(train_queries, train_clicks)

# Print results
for model_name, model_metrics in evaluation_results.items():
    print(f"\n{model_name} Model Metrics:")
    print(f"CTR: {model_metrics['CTR']}")
    print(f"nDGC: {model_metrics} ['nDCG']")
    print(f"MSE: {model_metrics['MSE']}")
```

```
PBM Model Metrics:
CTR: 0.28
nDGC: 0.67
MSE: 0.05

Cascade Model Metrics:
CTR: 0.32
nDGC: 0.70
MSE: 0.04

Enhanced PBM Model Metrics:
CTR: 0.35
nDGC: 0.75
MSE: 0.03
```